Distributed Persistence for Limited Devices

Philipp Bolliger and Marc Langheinrich

Inst. for Pervasive Computing
ETH Zurich, Switzerland
{bolligph,langhein}@inf.ethz.ch

Abstract. The problem of storing data both locally and remotely in a synchronized fashion is common in ubiquitous computing, where a mobile device (e.g., a tiny sensor node or a mobile phone) might produce a significant amount of data while having only limited storage capacity. While data management frameworks for small and mobile devices exist, they either have high system requirements (e.g., JavaSpaces) or do not focus on distributed storage issues (e.g., TinyDB). We have designed and implemented a lightweight storage system that allows the transparent use of both local *and* remote storage space, using identical semantics. It is based on a serialization framework that allows CLDC-enabled J2ME devices to transparently store Java objects in the (local) Record Management Store, or in a remote database via a wireless byte stream transmission. We have built a prototypical mobile application ontop of this framework and report on our experiences.

1 Introduction

As mobile phones become more and more common in everyday life [1], they will increasingly be used for more sophisticated applications and might eventually be the main computational device for a majority of the population. Energy constraints, however, will for the foreseeable future limit the computational resources on such devices, thus prompting the need for good programming frameworks that support the *Java Micro Edition* (J2ME), a Java distribution for resource-constrained devices, and in particular its *Connected Limited Device Configuration* (CLDC), which represents the smallest available language subset of Sun's Java, together with the *Mobile Information Device Profile* (MIDP), which includes support for mobile phones.

In contrast to the more powerful *Connected Device Configuration* (CDC), the usage of the CLDC implies some non-negligible language restrictions, such as:

- The lack of support for java.io.Serializable
- The Java Reflection API is not available
- It does not provide *Remote Method Invocation* (RMI)

Distributed persistent storage is a particular feature of mobile applications that need to store objects and/or data both locally and remotely. This becomes

important whenever collaboration among several mobile users is wanted: while information needs to be accessible off a central database for sharing purposes, a local, synchronized copy helps to reduce communication overhead and system responsiveness.

Persistent storage frameworks are a well researched area for which a large number of both commercial and free open-source implementations are available. Particularly in the context of distributed systems, a number of approaches attempt to take heterogeneous environments into account, such as Sun's JavaSpaces [2]. JavaSpaces has been developed as a part of Sun's JINI architecture, and glues processes together via a shared, network-accessible, and persistent object repository, called *space*. In order to interact, processes may read, write, or just "take" (i.e., read and delete) objects from the space. Nevertheless, JavaSpaces is not a relational or object oriented database and is thus not primarily designed as a data repository. More importantly, JavaSpaces explicitly uses reflection to obtain the field names, types, and class name of an entry, rendering it unsuitable in the reflection-less environment of CLDC enabled mobile phones. This also applies to other freely available distributed persistence frameworks, such as Hibernate [3], Cocobase [4], or Torque [5].

While the *FramePersist* [6] framework explicitly addresses CLDC-based Java environments on mobile phones, it lacks an actual implementation. Equally unavailable is the (planned) CLDC version of the *Data Persistence API*, which is part of the *Mobile Infrastructure* [7] of the German software giant SAP. This lack of distributed persistence frameworks that are able to operate under the resource- and language-constrained conditions of MIDP/CLDC-enabled mobile phones has prompted us to design and implement our own distributed persistence framework. The following sections will briefly describe a motivating scenario for persistent distributed storage, before introducing our design and outlining our implementation. We will conclude with our experiences and an outlook on subsequent steps.

2 Example Scenario

Let us consider a Java *MIDlet*-application running on a mobile phone, as shown in figure 1 below. It stores its user's business cards, i.e., his name, office address, and phone number, in its local *Record Management Store* (RMS) under a locally unique ID, e.g., `0x220E.F214`. At the same time, this information is replicated in the employee directory on a central server in order to allow distributed access by, e.g., colleagues and clients. Upon first upload to this database, a *globally* unique ID, e.g., `0x103B.0105`, is assigned to this card, and subsequently propagated to the user's mobile phone in order to override the previous local ID. Should the user decide to share his business card with another mobile user, he can now simply exchange its global ID, `0x103B.0105`, allowing the recipient to directly reference this data off the central database. By sharing this common reference, both mobile clients can have access to identical information, i.e., if the user updates his contact information, all other mobile clients can verify if their local

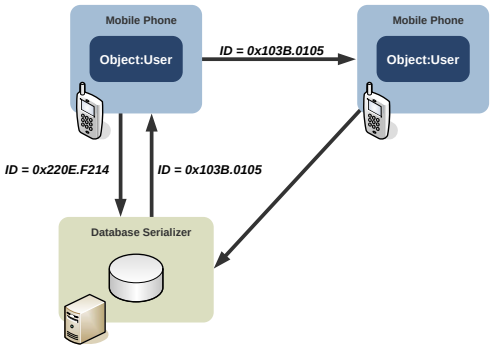

Fig. 1. Sample Use Case of the Remote Storage Service. A user's object is created locally (ID = `0x220E.F214`) and then stored on a central database. This prompts the creation of globally unique ID (`0x103B.0105`), which can be given to other mobile clients in order to allow them to reference (and sync to) the global copy.

copy is still current, or alternatively get informed by the central database that the data has been changed.

This (admittedly simple) example can easily be implemented in Java using existing persistent storage frameworks, such as the popular *JavaSpaces*. However, trying to port this (i.e., the remote storage and synchronization of objects) to any of today's mobile phones fails due to the lack of serialization support, reflection, and RMI capabilities in J2ME's CLDC.

3 Framework Design

In order to support the above functionality on today's mobile phone, we thus set out to create a minimal serialization framework that would allow us to transparently store data both in a phone's local RMS, as well as on a remote relational database management system (RDBMS). Instead of including direct RDBMS support on the resource-constrained mobile device, we opted to implement a generic byte stream serializer, which simply allows sending data to a corresponding module on the server side. The server component then includes a JDBC module that allows access to a local database. Figure 2 gives an overview of the entire framework, with the *Stream Serializer* being present in both server and mobile instances, and the *RMS Serializer* and *Database Serializer* being only available on the mobile client or the server, respectively.

The fundamental idea of our serialization API is that a *Serializable* object gets transformed into a stream, stored in a database or the RMS using the corresponding *Serializer*. The common serialization API abstracts the calls to the different serializers and hence simplifies their usage. All three serializers are able to serialize and deserialize *objects*, *object associations*, *object composition*, and even *class inheritance* (up to one level) with only one method call. To enable the serialization of *object collections*, a special vector object is available.

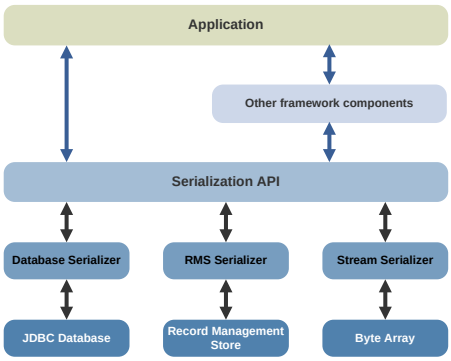

Fig. 2. Overview of the unified Serialization Framework. A *serialization API* provides transparent access to different *Serializers* for JDBC, RMS, and byte streams.

To distinguish the objects from each other, we use a numeric identifier, which is conceptually similar to the serialization API of J2SE [8]. The ID-space is partitioned into *local* (i.e., RMS) and *global* (i.e., database) identifiers, allowing a client to quickly differentiate replicated from local items. For practical purposes, global identifiers range from `0-0x1FFF.FFFF` (thus being identical to the RDBMs ID-space) and local identifier starting from `0x2000.0000`.

As RMS and RDBMS store data differently – RMS uses a record-oriented schema, compared to the table-oriented approach of RDBMS – we created a table-oriented RMS access scheme that allows us to reuse much of the RDBMS-serialization functionality. For this purpose, our system transparently organizes objects into *type stores* (according to their type) during serialization. For example, assume that a *User* object contains not only the user's name, but also a reference to a *Phone* object. During serialization, our system translates a single storage command into a number of separate *stores*, as shown in Figure 3, where the *Users*-store holds references to individual *User* objects, e.g., `User10001`, which in turn are represented by stores that contain references to other object-stores (e.g., `Phone10001`).

In order to support the use case as depicted in Figure 1, we need to combine the services of the different serializers to remotely store an object. This is done in the *Remote Storage* service, which allows mobile phones to instruct the server to remotely store an object in its database. Upon invocation, the service creates

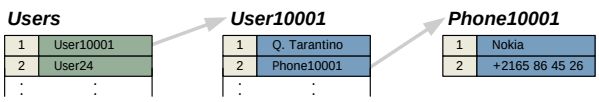

Fig. 3. Example view on the abstracted Record Management System. In order to allow code reuse from the Database Serializer, RMS records get transparently converted into several *type stores*, based on the contained object types.

a XML request message, serializes it using the Stream Serializer, and then sends it off to the server. Using tokenization and zip-compression as proposed in [9], we achieve average compression rates of 4:1 for these XML messages.

On the server side, an identical instance of the Stream Serializer receives the incoming message, reinstantiates the object, and stores it in its RDBMS using the Database Serializer. The object's local ID will be detected (being above `0x1FFF.FFFF`) and transparently converted to a new global ID, which will be sent back to the mobile client as a response to its storage request.

To enable the different and distributed applications to communicate with each other, we implemented a bidirectional, asynchronous communication mechanism based on messages. As messages are always sent directly between clients and server, or in between clients, we were able to omit routing information completely. Messages thus only consist of a sender and a type attribute, making the message definition relatively simple, yet still allowing for arbitrary complex messages. In order to minimize (monetary) communication costs, the client implementation dynamically chooses between SMS, MMS, and Bluetooth communication, depending on message size and service availability. Since contacting a mobile client directly from the server using IP-based services is difficult, we simply employ a polling mechanism on the mobile phone to fetch messages from the server.

In order to support a large number of concurrent client connections to the server, our server infrastructure can start multiple instances of a service, for example for handling object storage request. This is achieved by having every service dispatch a freely selectable number of worker threads that will process requests, and by increasing or decreasing the number of threads according to service load.

4 Experiences and Outlook

We have used our persistence framework to build a distributed query application that allows mobile phones to trace users objects (which are equipped with small identification tags), distribute useful context information related to these objects, and locate them when lost or misplaced [10]. The use of our framework has greatly simplified application development. Even though developing for the limited capabilities of the CLDC was challenging, we were able to successfully create a persistence framework that allows application programmers to store and load objects in and from the RMS, as well as in a remote RDBMS. As far as we know, our serialization framework is the fist that supports both JDBC databases and the Record Management System of MIDP. By combining these services, we were able to develop a remote storage service that, given an appropriate communication mechanism, makes the storage of data transparent for the application programmer.

Since the API of J2ME is limited, developing for this platform proved to be very time consuming, since we had to reimplement a non-trivial amount of standard Java functionality. This becomes especially relevant when developing

components that are intended to run on both the mobile phone and on the server. We have found it worthwhile to establish a suitable test environment that relieved us from continuously uploading programs to a real mobile phone.

As we strictly applied the OSGi specifications [11] during framework design, we also realized that the use of a component- and service-oriented architecture indeed has significant advantages. Since we were obliged to organize all component interaction through interfaces, we are now able to easily exchange individual components in the future. Furthermore, testing became much simpler, as the interfaces were already defined and allowed us to concentrate on the specification of the expected behavior. Every component was continually tested for correctness during development, using standard JUnit testing methods (for testing the remote storage services we needed to write our own extensions though).

One sobering experience was the wide variety of (supposedly standardized) CLDC implementations we encountered. Additionally, all of the different phones we tried featured a different version of the MIDP as well. As many of the needed functionalities, like Bluetooth or the Wireless Messaging API (WMA), are not part of the MIDP, these optional packages were often not available on a particular mobile phone. A mechanism to determine a phone's capabilities *before* compiling and installing a MIDlet would reduce development time and thus costs.

We are currently working on an additional security layer that should provide access rights and user groups. We also plan to introduce a notion of lifetime to the serialized data, in order to automatically delete outdated objects. Another interesting area of research that we want to look into is the implementation of transactions and concurrency control. Despite the often sever limitations of CLDC, we are confident that we can still significantly improve our distributed persistence framework for small, mobile devices.

References

1. Geser, H.: Towards a sociological theory of the mobile phone. (2004)
2. Sun Microsystems Inc.: JavaSpaces service specification. White paper (2003)
3. JBoss: Hibernate reference documentation. Manual (2005)
4. Thought, Inc.: CocoBase: Managing data in the enterprise. Tech report (2003)
5. Foundation, A.S.: http://db.apache.org/torque/ (2006) June 14, 2006.
6. Magalhaes, K.C.P., Carvalho, W.V., Lemos, F., Machado, J.C., Andrade, R.M.C.: FramePersist: An object persistence framework for mobile device applications. In: XIX Simposio Brasileiro de Banco de Dados, Brasilia, DF, Brasil (2004)
7. SAP AG: SAP mobile infrastructure: An open platform for enterprise mobility. White paper (2003)
8. Sun Microsystems, Inc.: Java object serialization specification. Tech report (2001)
9. Kochnev, D.S., Terekhov, A.A.: Surviving java for mobiles. IEEE Pervasive Computing **02**(2) (2003) 90–95
10. Frank, C., Roduner, C., Noda, C., Kellerer, W.: Query scoping for the sensor internet. In: IEEE International Conference on Pervasive Services (ICPS 2006), Lyon, France (2006)
11. Kriens, P., et al.: OSGi Service Platform Specification, Release 3. Tech report, The Open Services Gateway Initiative (2003)